CMPSC 465 Data Structures & Algorithms
Spring 2024 Mehrdad Mahdavi and David Koslicki

Worksheet 11

Wednesday, April 3, 2024

1. **Vertex Cover**

   (a) We can get this bound using a greedy algorithm similar to the one for Set Cover (We will formalize the connection between Vertex Cover and Set Cover later, using ”reduction”). For now, it is enough to see that edges are like the Set Cover’s ”items” and nodes cover some set of edges, so they are like the Set Cover’s ”sets”. We count the number of uncovered edges incident on every vertex, then pick the vertex that covers the most uncovered edges. After picking a vertex, remove all the edges it covers, decreasing the count of uncovered edges next to any adjacent vertices. Repeat the process until all edges have been covered. The same proof from class applies to this problem as well, since we kept the greedy algorithm the same. Following that proof shows that this greedy approach never uses more than $k\ln(|E|)$ vertices.

   (b) The greedy algorithm for this bound requires picking a maximal (cannot add any more) set of edges that share no endpoints, which is also called a Maximal Matching. Any such set of edges will do, it doesn’t need to be the largest possible. To build this set E', simply pick any edge, then remove all edges that share an endpoint with it, pick some remaining edge, and repeat until all edges have been chosen or removed. To create the Vertex Cover, we choose all vertices that are endpoints of some edge in E'.

   This will cover all the edges. The edges in the E' are trivially covered since their endpoints are included, but every edge not in the E' must have been removed for sharing an endpoint with an edge in E', and that endpoint has been included in our Vertex Cover. Therefore, all edges are covered.

   To show that this algorithm never uses more than $2k$ vertices, consider that the optimal Vertex Cover must cover every edge in E', and none of the edges in E' share any endpoints. Therefore, there must be at least $|E'|$ nodes in the optimal set cover: $k \geq |E'|$. Our algorithm returns a set of size $2|E'|$, which is at most $2k$. The worst-case graph for this greedy algorithm is one where no two edges share any endpoints. In this case, the greedy algorithm will use exactly $2k$ vertices, as it doesn’t get to remove any “free” edges that share an endpoint with an edge it has chosen.

2. **Worst-case Greedy Set Cover Instances.** This problem is much easier to visualize with a concrete example, so let’s first take $n = 64$ and the set of items $U = \{1,2,\dots,63,64\}$. Let $T_1 = \{1,3\dots,63\}$, and $T_2 = \{2,4,\dots,64\}$, so that picking T_1 and T_2 is the optimal answer. Now let $S_1 = \{1,2,\dots,32,33,34\}$, or 2 elements more than $64/2$. Let S_2 contain the next $64/4 = 16$ elements $\{35,36,\dots,49,50\}$, let S_3 contain the next $64/8 = 8$ elements $\{51,52\dots,57,58\}$, S_4

contain the next $64/16 = 4$ elements $\{59,60,61,62\}$, and S_5 contain the remaining 2 elements $\{63,64\}$. As you can see, these sets contain each number in U exactly once.

Our greedy algorithm will pick S_1,S_2,S_3,S_4,S_5 instead of T_1,T_2. This is because, at each step, the next set S_i covers slightly more uncovered elements than both T_1 and T_2. Our greedy algorithm ends up picking $\log(n)-1 = k-1$ sets, since $k = 6$ for $n = 64$. Now we generalize this result for all $n = 2^k$.

Consider the base set $U = \{1,2,3,\ldots,2^k\}$ for some $k \geq 2$. Let $T_1 = \{1,3,5,\ldots,2^k-1\}$ and $T_2 = \{2,4,6,\ldots,2^k\}$. These two sets comprise an optimal cover. We add sets $S_1,\ldots,S_{k-1}$ by defining $l_i = 2+\sum_{j=1}^{i} 2^{k-j}$ (take $l_0 = 0$) and letting $S_i = \{l_{i-1}+1,\ldots,l_i\}$. We think of S_i as covering a tiny bit more than a $1/2^i$-fraction of the universe, and in particular, S_1 is larger than both T_1 and T_2.

Since S_1 contains $2^{k-1}+2$ elements, it will be picked first. After the algorithm has picked $\{S_1,S_2,\ldots,S_i\}$, each of T_1 and T_2 covers $2^{k-i-1}-1$ new elements while S_{i+1} covers 2^{k-i-1} new elements. Hence, the algorithm picks the cover $S_1,\ldots,S_{k-1}$ containing $k-1 = \log n - 1$ sets.

3. **Maximum Non-Adjacent Sum.** Define S[i] as the maximum sum that can be achieved by a subsequence of $(a_1,...,a_i)$. For each integer a_i, we have two choices. If we choose a_i, S[i] = S[i-2] + a_i; if not, S[i] = S[i-1].

   The pseudo-code, which includes three steps, is as follows.

```
Initialization:
S[0] = 0
S[1] = $max\{0, a_1\}$

Iteration:
for i = 2···n:
    S[i] = $max\{S[i-1], S[i-2]+a_i\}$
endfor

Termination: S[n] gives the maximum sum that can be obtained.

Running time: O(n).
```

4. **Longest Common Subsequence.**
   Subproblems: Let us the define the following subproblems for all i,j such that $1 \leq i \leq n$, $1 \leq j \leq m$:

$$L(i,j) = \text{length of longest common subsequence between } x[1,\cdots,i] \text{ and } y[1,\cdots,j]$$

   Then, the solution to the original problem will be given by $L(n,m)$.

   Algorithm and Recursion: Initialize the dynamic program by setting $L(i,0) = 0$ and $L(0,j) = 0$ for all i,j. The recursion to compute the values $L(i,j)$ is the following:

$$L(i,j) = \max\{L(i-1,j), L(i,j-1), L(i-1,j-1)+\text{equal}(x_i,y_j)\}$$

where equal(x,y) is 1 if x and y are the same character and is 0 otherwise.

Correctness and Running Time: The recursion is correct as the alignment producing the longest common subsequence for $L(i,j)$ will have in its last position either a deletion, two characters matched (either equal or different characters) or an insertion. The running time is $O(mn)$ as there are mn subproblems, each of which takes $O(1)$ time.

5. File Reconstruction.

(a) *Subproblems:* Define an array of subproblems $S(i)$ for $0 \leq i \leq n$ where $S(i)$ is 1 if $s[1 \cdots i]$ is a sequence of valid words and is 0 otherwise.

Algorithm and Recursion: It is sufficient to initialize $S(0) = 1$ and update the values $S(i)$ in ascending order according to the recursion

$$S(i) = \max_{0 \leq j < i} \{ S(j) \mid \texttt{dict}(s[j+1 \cdots i]) = \texttt{true} \}$$

Then, the string s can be reconstructed as a sequence of valid words if and only if $S(n) = 1$.

Correctness and Running Time: Consider $s[1 \cdots i]$. If it is a sequence of valid words, there is a last word $s[j \cdots i]$, which is valid, and such that $S(j) = 1$ and the update will cause $S(i)$ to be set to 1. Otherwise, for any valid word $S[j \cdots i]$, $S(j)$ must be 0 and $S(i)$ will also be set to 0. This runs in time $O(n^2)$ as there are n subproblems, each of which takes time $O(n)$ to be updated with the solution obtained from smaller subproblems.

(b) Every time a $S(i)$ is updated to 1 keep track of the previous item $S(j)$ which caused the update of $S(i)$ because $s[j+1 \cdots i]$ was a valid word. At termination, if $S(n) = 1$, trace back the series of updates to recover the partition in words. This only adds a constant amount of work at each subproblem and a $O(n)$ time pass over the array at the end. Hence, the running time remains $O(n^2)$.